Verifying Declarative Netlog Protocols with Coq: a First Experiment

Yuxin Deng, Stéphane Grumbach, Jean-François Monin

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Verifying Declarative Netlog Protocols with Coq: a First Experiment

Yuxin Deng — Stéphane Grumbach — Jean-François Monin

N° 7511

Janvier 2011

Programs, Verification and Proofs

ISSN 0249-6399 ISRN INRIA/RR--7511--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
PARIS - ROCQUENCOURT

Verifying Declarative Netlog Protocols with Coq: a First Experiment

Yuxin Deng*, Stéphane Grumbach†, Jean-François Monin‡

Theme : Programs, Verification and Proofs
Algorithmics, Programming, Software and Architecture
Équipes-Projets FORMES et Netquest

Rapport de recherche n° 7511 — Janvier 2011 — 25 pages

Abstract: Declarative languages, such as recursive rule based languages, have been proposed to program distributed applications over networks. It has been shown that they simplify greatly the code, while still offering efficient distributed execution. In this report, we show that moreover they provide a promising approach to the verification of distributed protocols. We consider the Netlog language and use the Coq proof assistant. We first formalize the distributed computation model based on message passing with either synchronous or asynchronous behavior. We then see how the declarative rules of the protocols can be simply encoded in Coq and we develop the machine embedded on each node of the network which evaluates the rules. This framework enables us to formally verify distributed protocols, as shown on a concrete case study, a spanning tree construction in both the asynchronous and synchronous setting.

Key-words: formal proof, protocol, distributed algorithm, distributed computation model.

* BASICS, Shanghai Jiao Tong University, China
† LIAMA-INRIA, Netquest
‡ LIAMA-CNRS et Université Joseph Fourier Grenoble 1, FORMES

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Vérification de protocoles déclaratifs en Coq: une première expérience

Résumé : L'idée d'utiliser des langages déclaratifs, par exemple à base de règles récursives, a été proposée pour programmer des applications distribuées sur des réseaux. Il a été montré que cela simplifie grandement le code, sans sacrifier l'efficacité de l'exécution distribuée. Dans ce rapport, nous montrons qu'en outre ils constituent une approche prometteuse à la vérification de protocoles. Nous considérons le langage Netlog et utilisons l'assistant à la preuve Coq. Nous commençons par formaliser le modèle de calcul distribué par communication de message dans les deux variantes synchrone et asynchrone. Nous montrons ensuite un encodage simple en Coq des règles définissant un protocole et ainsi que de leur évaluation sur chaque nœud du réseau. Ce cadre permet de vérifier formellement des protocoles distribués, comme illustré sur une étude de cas concrète, la construction d'un arbre, tant dans le cas synchrone que dans le cas asynchrone.

Mots-clés : preuve formelle, protocole, algorithme distribué, modèle de calcul distribué.

1 Introduction

Programming distributed algorithms, such as networking protocols for instance, is a very complex task, which aims at solving global problems using local means, and requires to handle the concurrency of the processes, the delays or even the failure of the communication, as well as the limitations of both the hardware and the communication channels. Most distributed systems rely on algorithms invoking low level systems considerations. High-level abstractions have been proposed to facilitate the programming, based on graph relabeling [38, 4], rule-based languages such as [34, 23], functional languages such as Flask [37], as well as algebras for routing [22]. Rule-based languages provide a declarative programming framework which improves greatly the programmer's burden, with a code which is about two orders of magnitude shorter than standard programming languages, and has been shown to produce efficient algorithms in the case of various networking protocols in particular, by using methods developed in the field of databases for recursive languages à la Datalog [32].

In the present report, we show that these declarative languages for distributed programming provide a new approach to the verification of distributed programs, which can naturally deal with global properties, e.g. topological properties of a distributed data structure like a tree. To the best of our knowledge, such properties are hard to prove or even to state with usual techniques relying on la-belled transition systems and temporal logic, since they essentially focus on events and their ordering.

We choose to work with the Netlog language [23], a variant of Datalog recently proposed for programming distributed algorithms. The Netlog language relies on deductive rules of the form $head \leftarrow body$, which are installed on each node of the distributed system. The rules allow to derive new facts of the form "*head*", if their body is satisfied locally on the node. The facts derived might then be stored locally on the node or sent to other nodes in the network depending upon the rule. Netlog admits a semantics which is formally defined by distributed fix-point, which interleaves local computation on the nodes and communication between the nodes. On each node, a local round consists of a computation phase followed by a communication phase. During the computation phase, the program updates the local data and produces messages to send. During the communication phase, the router transmits the incoming messages to the program, and routes the outgoing messages.

Our objective is to develop a framework to formally verify properties of declarative distributed programs. As to formal verification, there are roughly two kinds of approaches: *model checking* and *theorem proving*. Model checking explores the state space of a system model exhaustively to see if a desirable property is satisfied. It is largely automated and generates a counterexample if the property doesn't hold. The state explosion problem limits the potential of model checkers for large systems. The basic idea of theorem proving is to translate a system's specification into a mathematical theory and then construct a proof of a theorem by generating the intermediate proof steps. Theorem proving can deal with large or even infinite state spaces by using proof principles such as induction and co-induction. In this report, we use Coq, which is a *proof assistant*, that is an interactive theorem prover, in which high level proof search commands construct formal proofs behind the scene, which are then mechanically verified. Using a proof assistant seems more relevant than model checking

here since the manipulation of data plays a key role. We develop a Coq library necessary for our purposes, including (i) the formalization of the distributed system; (ii) the modeling of the embedded machine evaluating the Netlog programs; (iii) the translation of the Netlog programs; as well as a formalization of graphs and trees suitable to our needs (respectively for communication networks and our case study).

Technically, we formalize a message passing model for distributed computation. To this effect, we introduce a general framework parameterized by a network topology and an abstract type for data. We then formalize appropriate notions for defining a global behavior in terms of local rounds, in a way such that synchronous and asynchronous behaviors are obtained from the same ingredients. This provides a transition relation between configurations (or global states), on which general definitions can be applied, for example, the co-inductive definition of a run and inductive or co-inductive definitions of temporal logic operators and associated proof principles.

In our Coq formalization, each body of a deductive rule is encoded in a systematic way by a tuple parameterized by a configuration, a node Id and the free variables of the body. In turn, each Netlog rule is formalized by a tuple which relates a configuration to a set of data representing updates to be performed atomically at a given node, if the corresponding body is satisfied.

As a proof of concept, we test the proposed framework on a concrete protocol for constructing spanning trees over connected graphs. It thus proceeds in rounds, in which one node (in the asynchronous model) or all nodes (in the synchronous model) perform some local computation and then exchange data with their neighbors before entering the next round. To show its correctness, the crucial ingredient is to formally prove the validity of the invariant that evolving from one round to another always produces a larger tree rooted at the same node. The protocol is shown to be correct for any finite connected graph. Furthermore, in the synchronous message passing model, we show that we get a distributed version of the classical breadth-first search (BFS) algorithm.

The report is organized as follows. In Section 2, we formalize the distributed computational model. Section 3 is devoted to the presentation of Netlog programs, and their translation in Coq. Section 4 provides the fix-point semantics of Netlog programs, while Section 5 presents the Coq formalization of the Netlog machine. Section 6 sketches the proofs of the correctness of the tree protocol. Section 7 discusses some related work, and finally we conclude in Section 8.

2 Distributed Computation Model

In this section we introduce a distributed computation model based on the message passing mechanism and then formalize it in Coq. A brief overview of Coq is delegated to Appendix A.

A *distributed system* relies on a communication network whose topology is given by a *directed connected graph* $\mathcal{G} = (V_{\mathcal{G}}, G)$, where $V_{\mathcal{G}}$ is the set of nodes, and G denotes the set of *communication links* between nodes. For many applications, we can also assume that the graph is symmetric, that is $G(\alpha, \beta) \Leftrightarrow G(\beta, \alpha)$.

Each node has a unique *identifier*, *Id*, taken from $1, 2, \cdots, n$, where $n \geq 2$ is the number of nodes, and distinct local ports for distinct links incident to it.

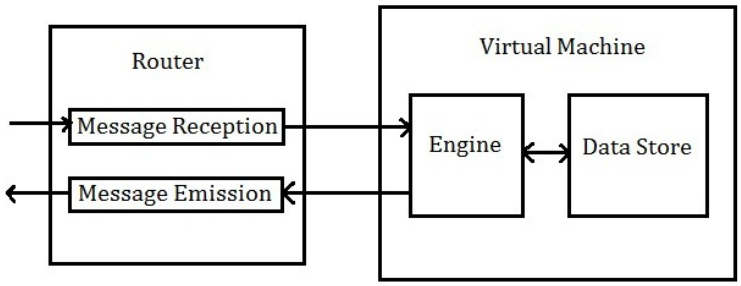

Figure 1: The node architecture

The control is fully distributed in the network, and there is no shared memory. In this high-level computation model, we abstract away detailed factors like node failures and lossy channels; if we were to formalize a more precise model, most of the data structures defined below would have to be refined.

All the nodes have the same architecture and the same behavior. As depicted in Figure 1, each node consists of three main components: (i) a router, handling the communication with the network; (ii) an engine, executing the local programs; and (iii) a local data store to maintain the information (data and programs) local to the node. It contains in particular the fragment of G, which relates a node to its neighbors. The router queues the incoming messages on the reception queue and the message to push produced by the engine on the emission queue.

We distinguish between *computation events*, performed in a node, and *communication events*, performed by nodes which cast their messages to their neighbors. On one node, a *computation phase* followed by a *communication phase* is called a *local round* of the distributed computation.

An *execution* is a sequence of alternating global configurations and rounds occurring on one node, in the case of an asynchronous system, or a sequence of alternating global configurations and rounds occurring simultaneously on each node, in the case of a synchronous system. In the latter case, the computation phase runs in parallel on all nodes, immediately followed by a parallel execution on all nodes of the corresponding communication phase.

The distributed computation model described above does not depend on Netlog. In our formalization, we just assume that the states at nodes have a type `local_data` which can evolve using simple set-theoretic operations such as union. Apart from that, the distributed computation model is quite standard. Note that it is suitable both for synchronous and asynchronous execution.

We represent finite sets by simple lists, the union is simply represented by list concatenation. This is a suitable choice as long as the only predicate we consider on lists is membership. In the sequel we use freely set-theoretic notation for list operations, e.g. $\emptyset$ for `nil` and $\in$ for `In`.

The detailed Coq script for formalizing the distributed computation model can be found in [17]. Here we just discuss a couple of key ingredients, especially for the representations of rounds in asynchronous and synchronous settings.

In the formal model, the graph is defined by a relation `edge` between nodes. This relation is itself defined by a function `neighbors` which provides the list of neighbors of a given node.

```
Variable neighbors : nat -> list nat.
Definition edge n m :=  m ∈ neighbors n.
```

We assume a type `local_data` for the set of facts stored on nodes as well as on communication links. This type is endowed with at least a value representing the *empty set* of facts and a binary function returning the *union* of two sets of facts. The union is used for describing incremental monotonic changes (see `local_round` and `communication` below). Non-monotonic changes such as removing facts are dealt with using an additional *set difference* function.

We also define the type `Bmsg` for "big messages", i.e. pairs (j, t) where j is a node Id and t a set of data to be transmitted to j. The global state of the system has the type `configuration` defined as follows.

```
Variable local_data: Set.
Variable empty_ld: local_data.                        (* notation ∅ld *)
Variable union_ld: local_data -> local_data -> local_data.  (* ∪ld *)

Definition Bmsg := nat * local_data.

Record configuration: Set:= mk_configuration {
  Cnode: nat -> local_data;
  Cedge: ∀ src dst: nat, edge src dst -> local_data
}.
```

Given a configuration c and a node Id j, the *data available at j in c* is either `Cnode` c j, or `Cedge` c e, where e is an edge from some node i to j (a more complete expression would be `Cedge` c i j e, but i and j can be easily deduced from the type of e; in what follows, we omit such implicit arguments, as is done in the formal Coq development).

A local round at node `loc` relates an actual configuration `pre` to a new configuration `mid` and a list `out` of big messages from `loc`. Furthermore, incoming edges are cleared. The new data `d` to be stored on `loc` is defined by a relation `new_stores` given as a parameter, and we assume that `d` depends only on the data available at `loc` in `pre`. Intuitively, the relation `new_stores` expresses that `d` consists of new facts derived from facts available at `loc` (see more details in Section 5). Similarly, `out` is defined by a relation `new_push` and satisfies similar requirements. Using relations rather than functions for `new_stores` and `new_push` deserves a special discussion provided in Section 3.

Formally, a local round is defined by using the following inference rule (by convention, for such rules, free variables should be read as universally quantified over the whole rule).

$$\frac{\begin{array}{l}\exists \texttt{d, new\_stores pre loc d} \wedge \texttt{Cnode mid loc} = \texttt{Cnode pre loc} \cup_{\texttt{ld}} \texttt{d} \\ \texttt{new\_push pre loc out} \\ \forall\ \texttt{src (e: edge src loc), Cedge mid e} = \emptyset_{\texttt{ld}}\end{array}}{\texttt{local\_round loc pre mid out}}$$

This definition expresses that a local round relating pre, mid and out at loc needs the following three components: a proof that a suitable d exists, a proof that pre

loc at out are related according to `new_push`, and a proof that for all nodes `src` related to `loc` by `edge e`, the data stored on `e` in configuration `mid` is empty.

For modeling asynchronous behaviors, we also need the notion of a trivial local round at `loc`, where the local data does not change and moreover incoming edges are not cleared either.

$$\frac{\texttt{Cnode mid loc} = \texttt{Cnode pre loc} \quad \forall\ \texttt{src (e: edge src loc), Cedge mid e} = \texttt{Cedge pre e}}{\texttt{no\_change\_at loc pre mid}}$$

A communication event at node `loc` specifies that the local data at `loc` does not change and that facts from `out` are appended on edges according to their destinations.

$$\frac{\texttt{Cnode post loc} = \texttt{Cnode mid loc} \quad \forall\ \texttt{dst (e: edge loc dst), Cedge post e} = \texttt{find dst out} \cup_{\texttt{ld}} \texttt{Cedge mid e}}{\texttt{communication loc mid post out}}$$

The function `find` returns the fact in `out` whose destination is `dst`. Note that none of the previous three definitions specifies completely the next configuration in function of the previous one. They rather constrain a relation between two consecutive configurations by specifying what should happen at a given location. Combining these definitions in various ways allows us to define a complete transition relation between two configurations, with either a synchronous or an asynchronous behavior.

$$\frac{\begin{array}{l}\texttt{loc: nat; mid: configuration; out: list Bmsg}\\ \texttt{local\_round loc pre mid out}\\ \forall\ \texttt{loc'}, \texttt{loc} \neq \texttt{loc'} \rightarrow \texttt{no\_change\_at loc' pre mid}\\ \texttt{communication loc mid post out}\\ \forall\ \texttt{loc'}, \texttt{loc} \neq \texttt{loc'} \rightarrow \texttt{communication loc' mid post} \emptyset\end{array}}{\texttt{async\_round pre post}}$$

An asynchronous round between two configurations `pre` and `post` is given by a node Id `loc`, an intermediate configuration `mid` and a list of big messages `out` such that there is a local round relating `pre`, `mid` and `out` on `loc` while no change occurs on `loc'` different from `loc`, and a communication relates `mid` and `out` to `post` on `loc` while nothing is communicated on `loc'` different from `loc`.

$$\frac{\texttt{mid: configuration; out: list Bmsg} \quad \forall\ \texttt{loc}, \exists\ \texttt{out}, \texttt{local\_round loc pre mid out} \wedge \texttt{communication loc mid post out}}{\texttt{sync\_round pre post}}$$

A synchronous round between two configurations `pre` and `post` is given by an intermediate configuration `mid` such that for all node Id `loc`, there exists a list of big messages `out` such that there is a local round relating `pre`, `mid` and `out` on `loc` and a communication relating `mid` and `out` to `post` on `loc`.

Now, given an arbitrary `trans` relation, which can be of the form `sync_round`, or `async_round`, or even of some alternative form, we can co-inductively define a run starting from a configuration. We have two cases: either there is a transition from configuration `pre` to configuration `post`, then any run from `post` yields a run from `pre`; or, in the opposite case, we have an empty run from `pre`. Altogether, a run from `pre` is either a finite sequence of transitions ended up with a configuration where no transition is available, or an infinite sequence of transitions, where consecutive configurations are related using `trans`.

```
CoInductive run: configuration -> Set :=
  | Rtrans: ∀ pre post, trans pre post -> run post -> run pre
  | Rterm: ∀ pre, (∀ post, ¬ trans pre post) -> run pre.
```

In order to prove properties on run, we define some temporal logic operators. For instance, in Section 6 we need a version of `always`, which is parameterized by a property `P` of configurations. In a more general setting, the parameter would be a property of runs.

$$\frac{\texttt{P pre} \quad \texttt{alw\_run P r}}{\texttt{alw\_run P (Rtrans pre post s r)}} \qquad \frac{\texttt{P pre}}{\texttt{alw\_run P (Rterm pre h)}}$$

It is clear that a property which holds initially and is invariant is always satisfied on a run. This fact is easily proved in the very general setting provided by Coq. However, to show that a property is invariant is usually much more difficult. We will see a concrete example in Section 6.

3 Declarative Netlog Protocols

We next introduce the Netlog language through simple protocols for construction of routes and trees. Only the main constructs are presented. A more thorough presentation of the language can be found in [23]. Netlog relies on Datalog-like recursive rules, of the form $head \leftarrow body$, which allow to derive the fact "*head*" whenever the "*body*" is satisfied. In contrast with other approaches to concurrency, the focus is not primarily on observing some output, but on the high-level data (i.e. Datalog facts) contained in nodes. Imagine, for example, a program for constructing routing tables. Such tables are intended to be used by other protocols and reasoning on their *contents* is more direct than considering events.

The rules of a program are applied in parallel, and the results are computed by iterating the rules over the local instance of the node, using facts either stored on the node or pushed by a neighbor. The following rules, for instance, define routes, stored in a relation $Route(Src, Hop, Dst)$, from the graph E, which from each source node, Src, and for each destination, Dst, gives the next hop, Hop, on the path to that destination.

Simple routes

$$\updownarrow Route(x, y, y) \quad \leftarrow \quad E(@x, y). \tag{1}$$
$$\updownarrow Route(x, y, z) \quad \leftarrow \quad E(@x, y); Route(y, u, z). \tag{2}$$

This program has the following effect, when applied on a node, say α. If there is a fact $E(\alpha, \beta)$, then $Route(\alpha, \beta, \beta)$ can be derived by Rule (1), and if there are facts $E(\alpha, \beta)$ and $Route(\beta, \gamma, \delta)$, then $Route(\alpha, \beta, \delta)$ can be derived by Rule (2). The symbol "@" in the literal $E(@x, y)$ in the body of the rules forces the variable x to be instantiated by the node Id, or, in other words, forces the rule to run on node x.

The Netlog programs are installed on each node, where they run concurrently. The facts deduced from rules can be stored on the node, on which the rules run, or sent to other nodes. The symbol $\updownarrow$ in the head of the rules means that the result has to be both stored on the local data store ($\downarrow$), and sent to neighbor nodes ($\uparrow$). In other words, Rule (1), for instance, is essentially

a combination of a store rule $\downarrow Route(x, y, y) \leftarrow E(@x, y)$ with a push rule $\uparrow Route(x, y, y) \leftarrow E(@x, y)$.

Computing the body of a rule is always performed on a given node. A fact is then considered to hold if and only if it occurs on this node (which includes an incoming edge). The language also contains *negation*; consistently, the negation of a fact holds if it does not occur on the node where the computation is performed. *Aggregation functions* can also be used in the head of rules to aggregate over all values satisfying the body of the rule. For instance, the function *min* will be used in the next example.

The following program, which constructs a spanning tree over a distributed system, relies on three relation symbols: E, $onST$, and ST; E represents the edge relation; and at any stage of the computation, $onST(\alpha)$ (respectively $ST(\alpha, \beta)$) hold iff the node α (respectively the edge (α, β)) is already on the intended tree.

Spanning Tree / BFS Protocol

$$\updownarrow onST(x) \quad \leftarrow \quad @x = 0. \tag{3}$$

$$\left.\begin{array}{r}\updownarrow onST(y)\\ \downarrow ST(min(x), y)\end{array}\right\} \quad \leftarrow \quad E(x, @y); onST(x); \neg onST(y). \tag{4}$$

Rule (3) runs on the unique node, say ρ, which satisfies the relation $\rho = 0$. It derives a fact $OnST(\rho)$, which is stored on ρ and sent to its neighbors. Rule (4) runs on the nodes ($@y$) at the border of the already computed tree. It chooses one parent (the one with minimal Id) to join the tree. Two facts are derived, which are both locally stored. The fact $onST(y)$ is pushed to all neighbors. Each fact $E(x, y)$ is assumed to be initially stored on node y. As no new fact $E(x, y)$ can be derived from Rules (3) and (4), the consistency of E with the physical edge relation holds forever. This algorithm aims at constructing suitable distributed relations $onST$ and ST. In Section 6, we will prove that they actually define a tree; moreover, in the synchronous setting they define a BFS tree.

The rules of Netlog programs can be encoded in Coq according to a systematic method which should be clear from the BFS example. To be more precise, we present here a specification of the semantics of Netlog rules, as relations between a configuration, a location and incremental changes of data, consistently with what is expected in a local round (see Section 2).

First we define an actual `local_data`, called `bfs_data`, needed in the BFS protocol. In Rules (3) and (4), there are three kinds of facts: $onST(x)$, $ST(x, y)$ and $E(x, y)$, with the first being unary and the last two being binary. So we introduce the types `unary` and `binary` to represent respectively sets (encoded by lists) of unary and binary facts.

```
Record bfs_data : Set := mk_bfs_data {
  onST : unary;
  E : binary;
  ST : binary}.
```

The pushing or emission of messages from a node to its neighbors always follows the same scheme which can be abstracted as a general pattern. Assuming an update `u` related to a configuration `pre` and location `loc` by a relation R given

as parameter, the list of `Bmsg` broadcasted from `loc` is made of all pairs $(\texttt{dst}, \texttt{u})$ such that `dst` is a neighbor of `loc`, as formalized by:

$$\frac{R\ \texttt{pre loc u}}{\texttt{push}\ R\ \texttt{pre loc}\ (\texttt{map}\ (\texttt{fun dst =>}\ (\texttt{dst}, \texttt{u}))\ (\texttt{neighbors loc}))}$$

This rule scheme converts a set of data `u` into a big message (`dst,u`), for every neighbor `dst` of node `loc`. These big messages will be pushed to appropriate destinations in the next communication phase. A common situation is when a fact has to be both stored and broadcasted, as happens with Rules (3) and (4) of the BFS example. Then R is then obtained from the corresponding store rule.

Rule (3) leads to two definitions in Coq, for both the store ($\downarrow$) and the push ($\uparrow$) consequences. The stored part is coded as follows:

$$\frac{\begin{array}{l}\texttt{ST upd} = \emptyset \\ \texttt{E upd} = \emptyset \\ \texttt{loc} = 0 \rightarrow \texttt{onST upd} = \{\texttt{loc}\} \qquad \texttt{loc} \neq 0 \rightarrow \texttt{onST upd} = \emptyset\end{array}}{\texttt{compute\_phase\_store\_onST\_initial pre loc upd}}$$

which says that for node 0, the update `upd` contains only the fact $onST(0)$, while other nodes produce no new fact at all. The push part is then coded by instantiating the relation R as `compute_phase_store_onST_initial` in the rule scheme mentioned above.

```
Definition compute_phase_push_onST_initial :=
   push compute_phase_store_onST_initial
```

For Rule (4), we first introduce the inductive definition literally translated from the body $E(x, @y); onST(x); \neg onST(y)$ as follows:

$$\frac{\texttt{in\_E cnf loc (x,y)} \qquad \texttt{in\_onST cnf loc x} \qquad \neg\texttt{in\_onST cnf loc y}}{\texttt{tree\_body cnf loc x y}}$$

The predicate `in_E cnf loc (x, y)` holds iff a fact `E (x, y)` is available in configuration `cnf` at node `loc`, similarly for the predicate `in_onST`. Both of them are specializations of the general predicate `inFact`, where `prj` is any projection from `local_data` to some `list X`.

$$\frac{\texttt{x} \in \texttt{prj(Cnode cnf loc)}}{\texttt{inFact prj cnf loc x}} \qquad \frac{\texttt{e: edge neighbor loc} \qquad \texttt{x} \in \texttt{prj(Cedge cnf e)}}{\texttt{inFact prj cnf loc x}}$$

According to these rules, the facts usable by a node either reside in its local data store or are pushed by a neighbor and so are kept in the reception queue of the node. We then introduce three definitions corresponding to the two derived facts, together with the two modes store and push for the first one.
$\downarrow onST(y) \leftarrow E(x, @y); onST(x); \neg onST(y)$ is coded by:

$$\frac{\begin{array}{l}\texttt{ST upd} = \emptyset \\ \texttt{E upd} = \emptyset \\ \forall\ \texttt{y},\ \texttt{y} \in \texttt{onST upd} \leftrightarrow ((\exists\ \texttt{x}, \texttt{tree\_body pre loc x y}) \wedge \texttt{y} = \texttt{loc})\end{array}}{\texttt{compute\_phase\_store\_onST\_tree pre loc upd}}$$

which means that if the body $E(x, @y); onST(x); \neg onST(y)$ is satisfied, then at node y the update `upd` only includes the fact $onST(y)$. On the other hand, the push rule $\uparrow onST(y) \leftarrow E(x, @y); onST(x); \neg onST(y)$ is coded by:

```
Definition compute_phase_push_onST_tree :=
   push compute_phase_store_onST_tree
```

The last rule $\downarrow ST(min(x), y) \leftarrow E(x, @y); onST(x); \neg onST(y)$ is coded by:

$$\frac{\begin{array}{l}\texttt{onST upd} = \emptyset \\ \texttt{E upd} = \emptyset \\ \textit{let } \texttt{P d} := \texttt{tree\_body pre loc x loc } \textit{in} \\ \forall\ \texttt{x}, \texttt{P x} \rightarrow \forall\ \texttt{m}, \texttt{is\_min pre m P} \rightarrow \texttt{ST upd} = \{\texttt{(m, loc)}\} \qquad \forall\ \texttt{x}, \neg\texttt{P x} \rightarrow \texttt{ST upd} = \emptyset\end{array}}{\texttt{compute\_phase\_store\_ST\_tree pre loc upd}}$$

Here `is_min pre m P` means that `m` is the smallest element `x` satisfying `P x`, but it does not ensure the existence of such an `x`. If the smallest element `m` does exist, then at node `loc` the update `upd` contains only the fact `(m,loc)`. Otherwise, `upd` would be empty.

One may wonder why we use relations everywhere instead of functions: relations are more general but less handy than functions especially in proofs. This matter of fact is indeed driven by the relational nature of Datalog, on which Netlog is based: facts may be derived or not according to the body of rules and available facts. Moreover, the sequential composition of functions with relations provides relations, hence even when trying to use functions in previous attempts, e.g. for `new_stores` and `new_push` in `local_round`, this eventually turned out to be not general enough.

4 The embedded Netlog machine

Netlog programs are running on the nodes of the network. They produce facts to store as well as facts to be sent to other nodes. They are evaluated by a machine which implements a precise semantics [23], which has been defined by fix-points in a way which is classical for rule-based languages such as Datalog. We present the semantics of a subset of the Netlog language of interest in this report. The language is restricted to a subset of the language constructs, and moreover, the rules are applied only once at each round, unlike in [23] where a local fix-point is computed at each local round.

We assume that all variables range over the sort $(\mathbb{N}, \leq)$, of the natural numbers. Given a finite set V of variables, a *valuation* over V is a mapping from V to $\mathbb{N}$. Let $\mathcal{V}(Var(r))$ be the set of valuations σ over $Var(r)$, the set of variables of a rule r.

Let an *instance* I be a finite interpretation of the relations of some schema S, which contains E, as well as some other relation symbols, such as *Route*, ST, etc. depending upon the programs. The *satisfaction* of the literals in the body of rule r by instance I and valuation σ is defined in a classical way, except for the universal literal, where: $(I, \sigma) \models \neg R(t_1, \ldots, -, \ldots, t_n)$ iff $R(\sigma(t_1), \ldots, C, \ldots, \sigma(t_n)) \notin I$, for any constant C. Assume the body of r, $body_r$, is $L_1; \ldots; L_\ell$. We have $(I, \sigma) \models body_r$ iff $(I, \sigma) \models L_i$, for each $i \in [1, \ell]$.

The valuation of the head, $head_r$, of rule r can now be defined. The aggregation functions, which can only occur in the head of rules, require some care. Let $Var^{\not{Agg}}(head_r)$ be the set of simple variables in the head, which are not arguments of aggregation functions. Let $\tau \in \mathcal{V}(Var^{\not{Agg}}(head_r))$. We extend τ to $\mathcal{V}(Var(r))$ with respect to interpretation I, as:

$$[\tau]_{I,r} = \{\sigma | \sigma \in \mathcal{V}(Var(r)), \sigma(x) = \tau(x), \text{ for all } x \in dom(\tau), \text{ and } (I, \sigma) \models body_r\}.$$

In the sequel, we assume that $[\tau]_{I,r} \neq \emptyset$. We define $\tau(head_r)$ as follows. First, if $head_r$ contains only simple variables and is of the form: $R(x_1, \ldots, x_n)$, then $\tau(head_r) = R(\tau(x_1), \ldots, \tau(x_n))$. More generally, if it is of the form:

$$R(x_1, \ldots, x_n, agg(y_1), \ldots, agg(y_m))$$

where agg denotes an aggregate function on multi-sets, and {{ }} denotes multi-set, then
$\tau(head_r) = R(\tau(x_1), \ldots, \tau(x_n), agg\{\{\sigma(y_1) | \sigma \in [\tau]_{I,r}\}\}, \ldots,$
$agg\{\{\sigma(y_m) | \sigma \in [\tau]_{I,r}\}\})$.

We can now define the set of positive consequences of a program P over an instance I, $\Delta_P^+(I)$, as well as the set of consumed facts, $\Delta_P^-(I)$.

$$\Delta_P^+(I) = \{\tau(head_r) | r \in P, \tau \in \mathcal{V}(Var^{\not{Agg}}(head_r)), [\tau]_{I,r} \neq \emptyset\}.$$
$$\Delta_P^-(I) = \{R(\sigma(t_1), \ldots, \sigma(t_n)) | r \in P, (I, \sigma) \models body_r,\ !R(t_1, \ldots, t_n) \text{ in } body_r\}.$$

It is not hard to see that $\Delta_P^-(I) \subseteq I$.

Let us now distinguish between $P_\downarrow$ the subset of store rules, and $P_\uparrow$ of push rules in P. Note that store-and-push rules belong to both sets.

We describe the behavior of a Netlog program on one node, say α. At each local round, it takes as input the local data on α and the data pushed by neighbor nodes to α, (`local_round`) and produces updated local data, and data to be pushed to each of its neighbor (`communication`). The node also forwards messages, that are not used in the local computation. Its interaction with the rest of the network is defined by the *communication function*: $\mathcal{R}^\alpha(\ell)$, which maps each local round ℓ to the set of incoming messages on node α at local round ℓ.

Note that at each local round, the router sorts the incoming messages into two sets $\mathcal{L}^\alpha(\ell)$, of *received facts*, and $\mathcal{F}^\alpha(\ell)$, of *messages to forward* to other nodes depending upon their destination: $\mathcal{L}^\alpha(\ell)$ contains the facts extracted from messages received from other nodes, with destination α, "*nbg*" (the neighbor of the sender), or "*all*" (the message is broadcasted to all nodes). $\mathcal{F}^\alpha(\ell)$ contains the messages received from other nodes, with a destination different from α or destination "*all*", which will be forwarded further to other nodes.

$\mathcal{F}^\alpha(0) = \emptyset$;
$\mathcal{F}^\alpha(\ell) = \{(dest, fact) \mid (dest, fact) \in \mathcal{R}^\alpha(\ell); dest \notin \{\alpha, nbg\}\}$;
$\mathcal{L}^\alpha(\ell) = \{fact \mid (dest, fact) \in \mathcal{R}^\alpha(\ell); dest \in \{\alpha, nbg, all\}\}$.

The computation relies on two *operators*, associated to program P, (i) for the data to store locally, $\Psi_P^\downarrow$, and (ii) for the data to push to other nodes, $\Psi_P^\uparrow$. They take as input the local instance I, and the received facts L.

- $\Psi_P^\downarrow(I, L) = \Delta_{P\downarrow}^+(I \cup L) \cup (I \backslash \Delta_P^-(I \cup L))$ defines the *store operator*, producing facts to store.
- $\Psi_P^\uparrow$ defines the *push operator*, producing messages to push:

$$\Psi_P^\uparrow(I \cup L) = \left\{ (dest, fact) \;\middle|\; \begin{array}{l} fact \in \Delta_{P\uparrow}^+(I \cup L); \text{ and} \\ \text{if } fact \text{ contains an address term } @\beta \\ \text{ or } @all, \text{then resp. } dest = \beta \text{ or } all; \\ \text{otherwise } dest = nbg. \end{array} \right\}$$

When a local round ℓ starts, the node α has a local instance $\mathcal{I}^\alpha(\ell)$, and has received facts $\mathcal{L}^\alpha(\ell)$, and messages to forward $\mathcal{F}^\alpha(\ell)$. It then starts its computation, and produces a new local instance $\mathcal{I}^\alpha(\ell+1) = \Psi_P^\downarrow(\mathcal{I}^\alpha(\ell), \mathcal{L}^\alpha(\ell))$, and a set of messages to push $\mathcal{P}^\alpha(\ell) = \Psi_P^\uparrow(\mathcal{I}^\alpha(\ell), \mathcal{L}^\alpha(\ell))$, which is then sorted by destination.

Let us now consider the communication between nodes. The messages to push are accumulated in $\mathcal{P}^\alpha(\ell)$. Their routes will be computed according to the knowledge node α has of the *Route* relation.

In the case of synchronous systems without failure, there is an explicit correspondence between the incoming and outgoing sets of messages.

Proposition 4.1 *[23] For synchronous systems without failure, we have for* $l \geq 0$*:*
$\mathcal{R}^\alpha(0) = \emptyset$,

$$\mathcal{R}^\alpha(\ell+1) = \left\{ (dest, fact) \;\middle|\; \begin{array}{l} \exists \beta \text{ s.t. } E(\beta,\alpha) \in I^\beta(\ell); \\ (dest, fact) \in \mathcal{P}^\beta(\ell); \text{and} \\ \text{if } dest \notin \{\alpha, nil, all\} \text{ then} \\ Route(\beta, \alpha, dest) \in I^\beta(\ell) \end{array} \right\}.$$

In the case of asynchronous systems, the function $\mathcal{R}^\alpha$ depends upon the distributed system, and in general might differ between two executions. The semantics is thus defined up to the system of communication function $\mathcal{R}^\alpha$ for each node α.

The semantics is defined as the local data store obtained on each node of the network, when no communication occurs anymore in the network. The termination is thus only implicit and globally defined. Clearly, programs can very well not terminate. It has been shown that subclasses of well-behaved Netlog programs terminate in polynomial time [23].

5 The Netlog Machine in Coq

Recall that in our computation model each node has an embedded Netlog machine, which implements a precise semantics given in Section 4. Our Coq formalization of the Netlog machine defines a specialization of the distributed computation model given in Section 2. This model is expressed in terms of two abstract relations, `new_stores` and `new_push`. The present section explains their definitions, according to the model given in the previous section.

We have seen in Section 3 how Netlog rules are represented in Coq in some examples. For store rules the type is `configuration -> nat -> local_data -> Prop` and for push rules `configuration -> nat -> list Bmsg -> Prop`. In order to manipulate them explicitly in a general setting, we assume two types `store_rule_name` and `push_rule_name`, as well as an appropriate semantics for each rule (a relation in `configuration -> nat -> local_data -> Prop`).

```
Variable store_rule_name push_rule_name : Set.

Variable store_sem_of : store_rule_name ->
     configuration -> nat -> local_data -> Prop.

Variable push_sem_of : push_rule_name ->
     configuration -> nat -> list Bmsg -> Prop.
```

```
Variable store_rule_order : list store_rule_name.
Variable push_rule_order : list push_rule_name.
```

Two simple auxiliary functions are needed; their definitions are standard (here Coq is used as a functional programming language):

- `merge`, which merges two lists of `Bmsg` into one;
- `find`, which returns the data associated with a given node Id in a list of `Bmsg`.

```
Fixpoint insert (loc: nat) (d: local_data) (l : list Bmsg) :
  list Bmsg :=
  match l with
  | nil => (loc, d) :: nil
  | (loc1, d1) :: l =>
      if beq_nat loc loc1 then (loc, union_ld d d1) :: l
      else (loc1, d1) :: insert loc d l
  end.

Fixpoint merge (l1 l2: list Bmsg) : list Bmsg :=
  match l1 with
  | nil => l2
  | (loc1, d1) :: l1 => insert loc1 d1 (merge l1 l2)
  end.

Fixpoint find (dst: nat) (l : list Bmsg) : local_data :=
  match l with
  | nil => empty_ld
  | (loc, d) :: l =>
      if beq_nat loc dst then d else find dst l
  end.
```

Then, from a configuration `pre` and a given list of `store_rule_name`, the future local data to be stored at node `loc` is inductively defined by:

$$\frac{}{\texttt{stores\_of\_list pre loc} \emptyset\ \emptyset_{\texttt{ld}}} \qquad \frac{\texttt{store\_sem\_of r pre loc updr} \quad \texttt{stores\_of\_list pre loc l updl}}{\texttt{stores\_of\_list pre loc (r::l) (updr} \cup_{\texttt{ld}} \texttt{updl)}}$$

Briefly speaking, the local data resulted from executing a list of store rules can be obtained by collecting all facts produced by each individual rule of the list. The list of `Bmsg` to be sent from `loc` in configuration `pre` is defined similarly by a predicate called `push_of_list`.

An equivalent definition is given below, which is more computational and sometimes more convenient to use. (Their equivalence is formally proved.)

```
Fixpoint stores_of_list
    (pre : configuration) (loc : nat)
    (l: list store_rule_name) : local_data -> Prop :=
  match l with
  | nil => fun u => u = empty_ld
  | r :: l =>
      fun u => ∃ updr, store_sem_of r pre updr loc ∧
        ∃ updl, stores_of_list pre loc l updl ∧ u = union_ld updr updl
  end.
```

Similarly, here is the list of `Bmsg` to be sent from `loc` in configuration `pre`:

```
Fixpoint push_of_list
    (pre : configuration) (loc : nat)
    (lp: list push_rule_name) : list Bmsg -> Prop :=
  match lp with
  | nil => fun l => l = nil
  | r :: lp =>
      fun l => ∃ lupdr, push_sem_of r pre lupdr loc ∧
        ∃ lupdlp, push_of_list pre loc lp lupdlp ∧ l = merge lupdr lupdlp
  end.
```

A local round is defined by just applying the previous definitions to a list `srl` of store rule names and a list `prl` of push rule names, where each element of `store_rule_name` occurs exactly once; similarly for `push_rule_name`. Each rule name acts as a trigger which entails the computation of the corresponding rule.

```
Variable srl : list store_rule_name.
Variable prl : list push_rule_name.

Definition new_stores_mach :=
  fun cnf loc d => stores_of_list cnf loc srl d.
Definition new_push_mach :=
  fun cnf loc l => push_of_list cnf loc prl l.

Definition local_round_mach := local_round new_stores_mach new_push_mach.
```

Example of application: BFS

For our running example about BFS, we provide the following rule names and semantics, which should be self-explanatory. The definition for push rule names is similar, thus omitted.

```
Inductive bfs_store_rule_name : Set :=
  store_onST_initial | store_onST_tree | store_ST_tree.

Definition bfs_store_sem_of (r : bfs_store_rule_name) :=
  match r with
  | store_onST_initial => compute_phase_store_onST_initial
  | store_onST_tree => compute_phase_store_onST_tree
  | store_ST_tree => compute_phase_store_ST_tree
  end.

Definition bfs_store_rule_order :=
  store_onST_tree :: store_ST_tree :: store_onST_initial :: nil.
```

The list `bfs_store_rule_order` is defined so that by iterating through it we model the executing of each store rule one by one in a node. The order of executing these rules are irrelevant because different orders always yield the same new facts during a round.
In the formal reasoning carried out in Section 6, we need to use the following form of assumption

```
bsr : bfs_synchronous_round cnf_pre cnf_post
```

To exploit this assumption, we first specialize `bsr` to a suitable location `x`. By expanding and destructing the definitions given in the current section we obtain an environment containing:

```
upd0, upd1, upd2 : bfs_data
Hupd0 : compute_phase_store_onST_tree pre loc upd0
Hupd1 : compute_phase_store_ST_tree pre loc upd1
Hupd2 : compute_phase_store_onST_initial pre loc upd2
eCnode : Cnode post loc = Cnode pre loc ∪ld upd0 ∪ld upd1 ∪ld upd2 ∪ld ∅ld
```

We are then in position to reason by cases on the facts contained in hack `Cnode post loc`, using knowledge specified by the representation of Netlog rules on the corresponding updates `upd0`, `upd1`, `upd2` or on the facts previously in `Cnode pre loc`.

```
Inductive bfs_push_rule_name : Set :=
  push_onST_initial | push_onST_tree (* | push_ST_tree *) .

Definition bfs_push_sem_of (r : bfs_push_rule_name) :
     configuration bfs_data -> list Bmsg -> nat -> Prop :=
  match r with
  | push_onST_initial => compute_phase_push_onST_initial
  | push_onST_tree => compute_phase_push_onST_tree
  end.

Definition bfs_push_rule_order :=
  push_onST_tree :: push_onST_initial :: nil.
```

6 Verification of a Tree Protocol

We conduct the verification in two settings: in the asynchronous case we prove that the previous protocol eventually constructs a spanning tree; in the synchronous case we prove that actually the protocol constructs a spanning tree by doing a breadth-first search in the network. We briefly sketch the first case study and then give more detailed discussion for the second one which involves a much more difficult proof.

In both cases we expect to show that the relation ST determines a spanning tree. However, this relation is distributed on the nodes and the Netlog protocol reacts only to a locally visible part of relations ST, $onST$ and E. The expected property is then stated in terms of the *union* of all ST facts available on the network.

In the asynchronous case, we have to check that when adding a new fact $ST(x, y)$ at some node loc then x is already on the tree while y is not yet. This is basically entailed by the body of the last rule, but additional properties are needed in order to ensure this rigorously. We use the following ones:

1. The E relation corresponds exactly to the edges.
2. An $onST(z)$ fact arriving at a node y is already stored on the sender x.
3. If an $onST(x)$ fact is stored on a node loc, then $x = loc$.
4. The $onST$ relation grows consistently with ST ($onST$ is actually the engine of the algorithm), and these two relations define a tree.

The first three properties are separately proved to be invariant. The last property is included in a predicate `is_tree`(o, s), which intuitively means that the

union of all $onST$ facts o and the union of all ST facts s are consistent and they define a tree. We prove that if at the beginning of a round the first three properties together with `is_tree`(o, s) hold, then at the end of the round `is_tree`(o, s) still holds. The conjunction of all the four properties then constitutes an invariant of the protocol. We check that the initial configuration generates a tree, then we have that in all configurations of any asynchronous run starting from the initial configuration, ST has the shape of a tree. This safety property is formalized in Coq (the script is available on-line [17]).

Liveness, i.e. each node is eventually a member of $onST$, can be easily proved, provided the graph is finite and connected, and a fairness property is assumed in order to discard uninteresting runs where an inactive node is continuously chosen for each local round, instead of another node having an enabled rule. The proof is by induction on the finite cardinality of the set $\overline{onST}$ of nodes which do not satisfy $onST$. If at some point of a run this set is non-empty, then at least one of its members is a neighbor of the current tree due to connectivity. By fairness, this node eventually performs a local round and is no longer in $\overline{onST}$. Formalizing such arguments involving liveness and fairness properties of infinite behaviors of distributed systems has already been done in Coq [18]. The issue of termination is simpler in the synchronous setting, since fairness is no more needed to remove fake stuttering steps.

For our second case study, the correctness proof of the BFS protocol, we prove that in the synchronous setting, the union of ST facts is the same as the one which would be computed by a centralized algorithm $\mathcal{C}$ running on the union of all facts. This is more subtle than one may expect at first sight, because decisions taken on a given node do not depend on the global relations $onST$ and ST, but only on the visible part, which is made of the locally stored facts and of the arriving messages. Moreover, the information contained in an arriving $onST$ fact is ephemeral: this fact is not itself stored locally (only its consequences are stored) and it will never be sent again. Indeed this information is available exactly at the right time. We therefore make a precise reasoning on the consistency of stored and transmitted facts with the computation that would be performed by the centralized algorithm $\mathcal{C}$.

Given a tree made of node Id's `li` and arcs `la`, it can be enlarged by extending it with new nodes and new arcs. Formally, $\mathcal{C}$ defines the arcs to be added by considering those nodes that are not in `li` but have neighbors in `li`. The newly added nodes and arcs are represented by the lists `new_lloc li` and `new_larc li`, respectively. The algorithms for `new_lloc` and `new_larc` are defined in Coq by simple functional programs.

Our main theorem states that a synchronous round in the distributed synchronous version corresponds to a step of computation performed by `new_larc`. Intuitively, this happens because the changes of $onST$ facts and ST facts go side by side with the addition of new nodes and arcs to a partially constructed tree. Here $onST$ facts play a prominent role in showing the correspondence.

Our invariant for the distributed BFS protocol involves several predicates, besides the first three properties given above for the asynchronous version. Some of them are quite natural. For instance, the predicates `correct_onST cnf li` and `complete_onST cnf li` together maintain that in configuration `cnf`, the union of $onST$ facts available on any node characterizes exactly the members of `li`, the list of node Id's in the tree constructed by $\mathcal{C}$. We have similar

predicates about ST facts. Some others are more technical. For example, the predicate `all_edges_good` asserts that, for any neighboring nodes x and y, if the fact $onST(x)$ is stored on x then y is receiving that fact or the fact $onST(y)$ is already stored on y. With those predicates at hand, we prove 9 useful propagation properties, e.g.,

```
Theorem propag_consistent_with :
  ∀ pre post, bfs_synchronous_round pre post ->
  E_corresponds_to_edges pre ->
  ∀ li la, correct_onST pre li -> complete_onST_node pre li ->
  all_edges_good pre ->
  consistent_with (global_ST pre) la ->
  consistent_with (global_ST post) (new_larc li ++ la).
```

Here, `E_corresponds_to_edges` describes Property 1 above, about the correspondence of relation E with edges, `global_ST` represents the union of all ST facts on the network, and `consistent_with` R `la` says that the binary relation R is equivalent to membership to the list `la`. Suppose the distributed system evolves from global configuration `pre` into configuration `post` during a synchronous round. This property says that if all the ST facts extracted from `pre` can be expressed by the list of arcs `la`, then so is the case for configuration `post` and the list `la` extended with new arcs in `new_larc li`. Its proof requires a careful use of the preconditions simultaneously available on several neighboring locations. The conjunction of the propagated properties gives rise to an invariant which is, as expected, a strengthening of the desired property.

Besides this global property, one may wonder whether `ST(x,y)` facts are located on relevant nodes, i.e. child nodes `y` in our case, so that this information could be used by a higher layer protocol for transmitting data towards the root. This is actually a simple consequence of Rules (3) and (4), since they ensure that `ST(x,y)` can only be stored on `y`. This is formally proved in our framework.

7 Related work

Declarative languages have been first used in the context of networks for sensor networks. TinyDB [36] and Cougar [16] offer the possibility to write distributed queries in SQL. More interestingly, recursive query languages have been used to express communication network algorithms such as routing protocols [34] and declarative overlays [33]. Distributed query languages thus provide new means to express complex network problems such as node discovery [3], route finding, path maintenance with quality of service [6], topology discovery, including physical topology [5], secure networking [2], or adaptive MANET routing [31].

Using formal techniques for verifying communication protocols is far from being a new idea. "Formal Description techniques" were developed by telecommunication laboratories from the beginning of the 1980s in order to specify and verify protocols to be standardized at ITU and ISO. Three languages came out. Two of them, Estelle and SDL, are based on asynchronous communicating automata's, while LOTOS is a process algebra based on CCS and CSP extended with algebraic data types [46]. Various verification tools, ranging from simulation to model checking were developed and applied to many case studies [48, 27, 45, 49, 42, 44, 26, 19, 21]. For the approach of verifying communication

protocols based on process algebras, the guiding idea is to model both the implementation and the specification of a protocol as processes in a process algebra, and then to employ automatic tools to check either the former is a refinement of the latter or they are behaviorally equivalent [14, 43]. For example, the Concurrency Workbench [14] is a verification tool based on CCS, FDR [43] is based on CSP [25], ProVerif [8] is based on the applied pi calculus [1], etc.

Other approaches include input/output automata's [35], or Unity and TLA, which combine temporal logic and transition-based specification [12, 28]. Note that the two latter got support from proof assistant technology [41, 24, 13, 29].

A common feature to these approaches is their focus on control, in particular how to deal with behaviors in a distributed framework. Typical issues include non-determinism, deadlock freedom, stuttering, fairness, distributed consensus and, more recently, mobility. Data is generally considered as an abstract object not really related to the behavior. This is relevant for many low-level protocols, such as transport protocols. However, this does not suit the needs of applications which aim at building up a distributed global information, such as topological information on the network (in a physical or virtual sense), as in routing tables, for example. To our knowledge, such problems have not been attacked by means of the above mentioned approaches. An explanation may be that the pieces of data involved in distributed computations are embedded in different components of the global configuration, and the previous formalisms make it difficult to isolate them or to consider them as a whole in reasoning. A clear feature of the current report, compared with those previous approaches, is the emphasis on manipulating data in formal reasoning, which also drove us to use Coq as the verification tool. Beyond formal verification of distributed protocols, Coq has been successfully applied to ensure reliability of hardware and software systems in various fields, such as multiplier circuits [40], concurrent communication protocols [20], self-stabilizing population protocols [18], Local Computation Systems [11, 9, 10], devices for broadband protocols [39], and compilers [30] to name a few.

Closely related to our work is [47], where a declarative network verifier (DNV) was presented which maps specifications written in the Network Datalog query language into logical axioms which can be used in theorem provers like PVS to validate protocol correctness. The reasoning based on DNV is for Datalog specifications of (eventually distributed) algorithms, but not for distributed versions of Datalog such as the one provided by Netlog. In other words, it only considers the highly abstract centralized behavior of a network. In contrast, our development in this report is to reason about the distributed behavior of individual nodes which together yield some expected global behavior of the whole network. Therefore, we need to involve deep subtleties on message passing and derivation of local facts, which are all absent in [47].

8 Conclusion

We developed a Coq library for verifying declarative protocols expressed in a rule-based language.

This library includes the formalization of the distributed computation environment with the communication network. Importantly, we have shown that both the synchronous and the asynchronous models of communication can be

formalized in very similar ways. This library includes the formalization of the distributed computation environment with the communication network, where both the synchronous and the asynchronous models of communication are formalized in very similar ways. The library also includes the embedded machine which evaluates the Netlog programs on each node. The Netlog programs are translated into straightforward Coq definitions. As a preliminary result we proved a topological property of a distributed data structure – a tree – constructed by a simple but subtle program. To our knowledge, such properties are difficult to handle in other approaches to the verification of distributed programs. From this experiment, we are in position to define a deep embedding for systematically deriving Coq encodings from the abstract syntax of Netlog rules, as well as dedicated tactics for handling tedious steps specific to Netlog, and then plan to further verify declarative protocols for routing, election, naming, and other fundamental distributed problems.

References

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proc. POPL'01*, volume 36, pages 104–115. ACM, 2001.

[2] M. Abadi and B. T. Loo. Towards a declarative language and system for secure networking. In *Proc. NETB'07*, pages 1–6. USENIX Association, 2007.

[3] G. Alonso, E. Kranakis, C. Sawchuk, R. Wattenhofer, and P. Widmayer. Probabilistic Protocols for Node Discovery in Ad Hoc Multi-channel Broadcast Networks. In *Proc. ADHOC-NOW'03*, 2003.

[4] M. Bauderon, Y. Métivier, M. Mosbah, and A. Sellami. Graph Relabelling Systems: a Tool for Encoding, Proving, Studying and Visualizing - Distributed Algorithms. *ENTCS*, 51, 2001.

[5] Y. Bejerano, Y. Breitbart, M. N. Garofalakis, and R. Rastogi. Physical Topology Discovery for Large Multi-Subnet Networks. In *Proc. INFOCOM'03*, 2003.

[6] Y. Bejerano, Y. Breitbart, A. Orda, R. Rastogi, and A. Sprintson. Algorithms for computing QoS paths with restoration. *IEEE/ACM Trans. Netw.*, 13(3), 2005.

[7] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[8] B. Blanchet. Automatic Verification of Correspondences for Security Protocols. *Journal of Computer Security*, 17(4):363–434, 2009.

[9] P. Castéran and V. Filou. Tâches, types et tactiques pour les systèmes de calculs locaux. In *Journées Francophones des Langages Applicatifs*. Inria, 2010.

[10] P. Castéran and V. Filou. Tasks, Types and Tactics for Local Computation Systems. *Studia Informatica Universalis*, 2011. To appear, extended version of [9].

[11] P. Castéran, V. Filou, and M. Mosbah. Certifying Distributed Algorithms by Embedding Local Computation Systems in the Coq Proof Assistant. In *Symbolic Computation in Software Science (SCSS'09)*, 2009.

[12] K. M. Chandy. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., 1988.

[13] B. Chetali. Formal Verification of Concurrent Programs Using the Larch Prover. *IEEE Transactions on Software Engineering*, 24:46–62, 1998.

[14] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrency systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, 1993.

[15] Coq. The Coq Proof Assistant Reference Manual. `http://coq.inria.fr/V8.1pl3/refman/index.html`.

[16] A. J. Demers, J. Gehrke, R. Rajaraman, A. Trigoni, and Y. Yao. The Cougar Project: a work-in-progress report. *SIGMOD Record*, 32(4):53–59, 2003.

[17] Y. Deng, S. Grumbach, and J.-F. Monin. Coq script for Netlog protocols. `http://www-verimag.imag.fr/~monin/Proof/NetlogCoq/netlogcoq.tar.gz`.

[18] Y. Deng and J.-F. Monin. Verifying Self-stabilizing Population Protocols with Coq. In *Proc. TASE'09*, pages 201–208. IEEE Computer Society, 2009.

[19] J.-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A toolbox for the verification of LOTOS programs. In *Proc. ICSE '92*, pages 246–259. ACM, 1992.

[20] E. Giménez. *A Calculus of Infinite Constructions and its application to the verification of communicating systems*. PhD thesis, ENS Lyon, 1996.

[21] R. Gotzhein and J. Bredereke, editors. *Formal Description Techniques IX: Theory, application and tools, IFIP TC6 WG6.1*, volume 69 of *IFIP Conference Proceedings*. Chapman & Hall, 1996.

[22] T. G. Griffin and J. L. Sobrinho. Metarouting. In *Proc. ACM SIGCOMM'05*, 2005.

[23] S. Grumbach and F. Wang. Netlog, a Rule-Based Language for Distributed Programming. In *Proc. PADL'10*, volume 5937 of *LNCS*, pages 88–103, 2010.

[24] B. Heyd and P. Crégut. A Modular Coding of UNITY in COQ. In *Proc. TPHOLs'96*, pages 251–266. Springer, 1996.

[25] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[26] C. Jard, J. F. Monin, and R. Groz. Development of Veda, a Prototyping Tool for Distributed Algorithms. *IEEE Trans. Softw. Eng.*, 14(3):339–352, 1988.

[27] C. Kirkwood and M. Thomas. Experiences with specification and verification in LOTOS: a report on two case studies. In *Proc. WIFT '95*, page 159. IEEE Computer Society, 1995.

[28] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[29] T. Långbacka. A HOL Formalisation of the Temporal Logic of Actions. In *Proc. TPHOL'94*, pages 332–345. Springer, 1994.

[30] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. POPL'06*, pages 42–54. ACM, 2006.

[31] C. Liu, Y. Mao, M. Oprea, P. Basu, and B. T. Loo. A declarative perspective on adaptive manet routing. In *Proc. PRESTO '08*, pages 63–68. ACM, 2008.

[32] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *Proc. ACM SIGMOD'06*, 2006.

[33] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Proc. SOSP'05*, 2005.

[34] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *Proc. ACM SIGCOMM '05*, 2005.

[35] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[36] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1), 2005.

[37] G. Mainland, G. Morrisett, and M. Welsh. Flask: staged functional programming for sensor networks. In *Proc. ICFP'08*, 2008.

[38] Y. Métivier and E. Sopena. Graph Relabelling Systems: A General Overview. *Computers and Artificial Intelligence*, 16(2), 1997.

[39] J.-F. Monin. Proving a real time algorithm for ATM in Coq. In *Types for Proofs and Programs*, volume 1512 of *LNCS*, pages 277–293. Springer, 1998.

[40] C. Paulin-Mohring. Circuits as Streams in Coq: Verification of a Sequential Multiplier. In *Proc. TYPES'96*, volume 1158 of *LNCS*, pages 216–230. Springer, 1996.

[41] L. C. Paulson. Mechanizing UNITY in Isabelle. *ACM Trans. Comput. Logic*, 1(1):3–32, 2000.

[42] F. Regensburger and A. Barnard. Formal Verification of SDL Systems at the Siemens Mobile Phone Department. In *Proc. TACAS '98*, pages 439–455. Springer, 1998.

[43] A. W. Roscoe. *Model-checking CSP*, chapter 21. Prentice-Hall, 1994.

[44] S. M. Shahrier and R. M. Jenevein. SDL Specification and Verification of a Distributed Access Generic opticalNetwork Interface for SMDS Networks. Technical report, University of Texas at Austin, 1997.

[45] M. Törö, J. Zhu, and V. C. M. Leung. SDL specification and verification of universal personal computing: with Object GEODE. In *Proc. FORTE XI / PSTV XVIII '98*, pages 267–282. Kluwer, B.V., 1998.

[46] K. J. Turner. *Using Formal Description Techniques: An Introduction to Estelle, Lotos, and SDL.* John Wiley & Sons, Inc., 1993.

[47] A. Wang, P. Basu, B. T. Loo, and O. Sokolsky. Declarative Network Verification. In *Proc. PADL '09*, pages 61–75. Springer, 2009.

[48] J.-P. Wu and S. T. Chanson. Translation from LOTOS and Estelle Specifications to Extended Transition System and its Verification. In *Proc. FORTE '89*, pages 533–549. North-Holland Publishing Co., 1990.

[49] W. Zhang. Applying SDL Specifications and Tools to the Verification of Procedures. In *Proc. SDL '01*, pages 421–438. Springer, 2001.

A Formal Verification with Coq

Coq is one of the most popular proof assistants for formal verification. It is based on a constructive type theoretic setting, called the *Calculus of (co-)Inductive Constructions* (CIC), which can be summarized both as a polymorphic typed lambda-calculus enriched with universes, inductive and co-inductive types and a language for describing mathematical definitions and proofs [15, 7]. These two aspects are actually related thanks to the well-known Curry-Howard-De Bruijn isomorphism, which maps propositions to types and proofs to functional objects or strongly normalizing programs.

Let us illustrate some concepts and the syntax of Coq by a few examples. One of the most commonly used data structure is `list`. Let `A` be a type, a list whose elements are of type `A` can be inductively defined, with the usual constructors `nil` and `cons`:

```
Inductive list : Type :=
  | nil : list
  | cons : A -> list -> list.
```

Total recursive functional programs can be defined for lists. For example, the `In` predicate defined below checks if `a` occurs in the list `l`.

```
Fixpoint In (a:A) (l:list) {struct l} : Prop :=
    match l with
      | nil => False
      | b :: m => b = a \/ In a m
    end.
```

In order to ensure termination, a structurally decreasing argument is specified by `struct l`. Here we meet the realm of propositions `Prop`. A predicate over natural numbers for instance has type `nat -> Prop`. Given such a predicate `P`, a proof `p0` of `P 0` and a proof `step` of `for all n, P n -> P (S n)`, we can construct a proof of `P n` for all natural numbers `n`, using the following functional (primitive recursive) program:

```
Fixpoint natind (n:nat) {struct n}: P n := match n return (P n) with
   | 0 => p0
   | S q => step q (natind q)
  end.
```

The type of `natind` is `for all n, P n`, that is a *dependent type*, since the type of the result depends on the value of the argument; `step`, seen as a function from numbers `n` and proofs of `P n` and returning a proof of `P (S n)`, has a slightly more complex dependent type. In the match construct itself, the type of the result depends on the branch – it could be `P 0` or `P (S q)` for some `q`. Abstracting `P`, `p0` and `step` in `natind` yields a proof of the usual induction principle over natural numbers. As a function, it illustrates some important features of the type theory of Coq: polymorphism, inductive and dependent types.

Other constructs used in this report, such as *records*, are special cases of inductive types (i.e with only one constructor; *fields* are just projections). When defining inductive types, dependent types can also be used for constructors. It is especially convenient for formalizing algebraic structures (a carrier, operations and algebraic laws) and we use them extensively in the sequel. For example, graphs can be defined below, with two fields: `Vert` for vertexes and `Edge` for edges.

```
Record Graph : Type := mkGraph {
  Vert : Type;                  (* vertices *)
  Edge : Vert -> Vert -> Prop   (* edges *)
}.
```

Like functions, relations are widely used mathematical concepts in formal verification. As an example, let `R` be a binary relation over natural numbers. Its transitive closure can be defined as follows.

```
Inductive TC (R: nat->nat->Prop): nat -> nat -> Prop :=
  | TC0 : forall x y, R x y -> TC R x y
  | TCrec : forall x y z, R x z -> TC R z y -> TC R x y.
```

Some frequently used types such as `nat` and `list` are available in the standard library of Coq; by importing relevant packages, we can directly use the operations (e.g. `In` seen above) associated with lists. However, many other types are not in the library, and in this case they need to be defined from

scratch. We have seen `Graph` for graphs above. We next consider trees which will be used to reason about algorithms for constructing spanning trees on connected graphs. We define abstract trees inductively as follows (from now on, we use the notation ∀ instead of `for all` in order to save space; types of variables which can be easily inferred from the context are not explicitly given):

```
Variable Carrier : Set.
Inductive tree : list Carrier -> list (Carrier * Carrier) -> Prop :=
  | root : ∀ x: Carrier, tree (x :: nil) nil
  | leaf : ∀ lv le, tree lv le ->
            ∀ x y: Carrier, In x lv -> ¬ In y lv ->
            tree (y :: lv) ((x,y) :: le).
```

A tree is built upon a set of vertexes (represented by a list `lv`) and a set of edges (represented by a list `le` of pairs) by:

- the tree reduced to its root and no edge;
- from a tree upon `lv` and `le`, adding a new pendant vertex y and edge (x, y) such that $x \in$ `lv`, $y \notin$ `lv`, we obtain a tree upon `lv` extended with y and `le` extended with (x, y).

Altogether, the features of Coq allow us to formalize mathematical theories in a typed and precise but still very general setting. Coq offers an environment where users can state mathematical definitions using types, concrete objects, functions over them, and then interactively prove theorems. Obvious proof steps are automated, but clever ones, e.g. inductive arguments or intermediate sub-goals, require user interactions.

Contents

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399